**正则表达式**

**<1> 正则表达式概述**

正则表达式，又称正规表示式、正规表示法、正规表达式、规则表达式、常规表示法（英语：Regular Expression，在代码中常简写为regex、regexp或RE），是计算机科学的一个概念。正则表达式使用单个字符串来描述、匹配一系列匹配某个句法规则的字符串。在很多文本编辑器里，正则表达式通常被用来检索、替换那些匹配某个模式的文本。Regular Expression的“Regular”一般被译为“正则”、“正规”、“常规”。此处的“Regular”即是“规则”、“规律”的意思，Regular Expression即“描述某种规则的表达式”之意。

**<2>** **re模块操作**

在Python中需要通过正则表达式对字符串进行匹配的时候，可以使用一个模块，名字为re

**1. re模块的使用过程**

#coding=utf-8

# 导入re模块

import re

# 使用match方法进行匹配操作

result = re.match(正则表达式,要匹配的字符串)

# 如果上一步匹配到数据的话，可以使用group方法来提取数据

result.group()

**re.match是用来进行正则匹配检查的方法，若字符串匹配正则表达式，则match方法返回匹配对象（Match Object），否则返回None（注意不是空字符串""）。匹配对象Macth Object具有group方法，用来返回字符串的匹配部分**

**2. re模块示例(匹配以itcast开头的语句)**

#coding=utf-8

import re

result = re.match("itcast","itcast.cn")

result.group()

**3. 说明**

re.match() 能够匹配出以xxx开头的字符串

**<3> 表示字符**

| **字符** | **功能** |
| --- | --- |
| . | 匹配任意1个字符（除了\n） |
| [ ] | 匹配[ ]中列举的字符 |
| \d | 匹配数字，即0-9 |
| \D | 匹配非数字，即不是数字 |
| \s | 匹配空白，即 空格，tab键 |
| \S | 匹配非空白 |
| \w | 匹配单词字符，即a-z、A-Z、0-9、\_ |
| \W | 匹配非单词字符 |

**<4> 原始字符串**

**Python中字符串前面加上 r 表示原生字符串，**

与大多数编程语言相同，正则表达式里使用"\"作为转义字符，这就可能造成反斜杠困扰。假如你需要匹配文本中的字符"\"，那么使用编程语言表示的正则表达式里将需要4个反斜杠"\\"：前两个和后两个分别用于在编程语言里转义成反斜杠，转换成两个反斜杠后再在正则表达式里转义成一个反斜杠。

Python里的原生字符串很好地解决了这个问题，有了原始字符串，你再也不用担心是不是漏写了反斜杠，写出来的表达式也更直观。

>>> ret = re.match(r"c:\\a",mm).group()

>>> print(ret)

c:\a

**<5> 表示数量**

| **字符** | **功能** |
| --- | --- |
| \* | 匹配前一个字符出现0次或者无限次，即可有可无 |
| + | 匹配前一个字符出现1次或者无限次，即至少有1次 |
| ? | 匹配前一个字符出现1次或者0次，即要么有1次，要么没有 |
| {m} | 匹配前一个字符出现m次 |
| {m,} | 匹配前一个字符至少出现m次 |
| {m,n} | 匹配前一个字符出现从m到n次 |

| **字符** | **功能** |
| --- | --- |
| ^ | 匹配字符串开头 |
| $ | 匹配字符串结尾 |
| \b | 匹配一个单词的边界 |
| \B | 匹配非单词边界 |

**<6> 表示边界**

**<7> 匹配分组**

| **字符** | **功能** |
| --- | --- |
| | | 匹配左右任意一个表达式 |
| (ab) | 将括号中字符作为一个分组 |
| \num | 引用分组num匹配到的字符串 |
| (?P<name>) | 分组起别名 |
| (?P=name) | 引用别名为name分组匹配到的字符串 |

**<8> re模块的高级用法**

**search （搜索匹配一次）**

需求：匹配出文章阅读的次数

ret = re.search(r"\d+", "阅读次数为 9999")

ret.group()

**findall （搜索匹配所有）**

需求：统计出python、c、c++相应文章阅读的次数

ret = re.findall(r"\d+", "python = 9999, c = 7890, c++ = 12345")

print ret

**sub （将匹配到的数据进行替换）**

需求：将匹配到的阅读次数加1

方法1

ret = re.sub(r"\d+", '998', "python = 997")

print ret

方法2

def add(temp):

strNum = temp.group()

num = int(strNum) + 1

return str(num)

ret = re.sub(r"\d+", add, "python = 997")

print ret

ret = re.sub(r"\d+", add, "python = 99")

print ret

**split （根据匹配进行切割字符串，并返回一个列表）**

需求：切割字符串“info:xiaoZhang 33 shandong”

ret = re.split(r":| ","info:xiaoZhang 33 shandong")

print ret

**compile （创建匹配模式对象）**

>>> **phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')**

>>> **mo = phoneNumRegex.search('My number is 415-555-4242.')**

>>> **print('Phone number found: ' + mo.group())**

Phone number found: 415-555-4242

**<9> python贪婪和非贪婪**

Python里数量词默认是贪婪的（在少数语言里也可能是默认非贪婪），总是尝试匹配尽可能多的字符；非贪婪则相反，总是尝试匹配尽可能少的字符。

在"\*","?","+","{m,n}"后面加上？，使贪婪变成非贪婪。

>>> s="This is a number 234-235-22-423"

>>> r=re.match(".+(\d+-\d+-\d+-\d+)",s)

>>> r.group(1)

'4-235-22-423'

>>> r=re.match(".+?(\d+-\d+-\d+-\d+)",s)

>>> r.group(1)

'234-235-22-423'

正则表达式模式中使用到通配字，那它在从左到右的顺序求值时，会尽量“抓取”满足匹配最长字符串，在我们上面的例子里面，“.+”会从字符串的启始处抓取满足模式的最长字符，其中包括我们想得到的第一个整型字段的中的大部分，“\d+”只需一位字符就可以匹配，所以它匹配了数字“4”，而“.+”则匹配了从字符串起始到这个第一位数字4之前的所有字符。

解决方式：非贪婪操作符“？”，这个操作符可以用在"\*","+","?"的后面，要求正则匹配的越少越好。

>>> re.match(r"aa(\d+)","aa2343ddd").group(1)

'2343'

>>> re.match(r"aa(\d+?)","aa2343ddd").group(1)

'2'

>>> re.match(r"aa(\d+)ddd","aa2343ddd").group(1)

'2343'

>>> re.match(r"aa(\d+?)ddd","aa2343ddd").group(1)

'2343'

**通过在字符分类的左方括号 [ ] 后加上一个插入字符（^），就可以得到“非字符类”。**

非字符类将匹配不在这个字符类中的所有字符

>>> **consonantRegex = re.compile(r'[^aeiouAEIOU]')**

>>> **consonantRegex.findall('RoboCop eats baby food. BABY FOOD.')**

['R', 'b', 'c', 'p', ' ', 't', 's', ' ', 'b', 'b', 'y', ' ', 'f', 'd', '.', '

', 'B', 'B', 'Y', ' ', 'F', 'D', '.']

**非字点-星将匹配除换行外的所有字符。通过传入re.DOTALL 作为re.compile()的第二个参数，可以让句点字符匹配所有字符，包括换行字符。**

>>> **noNewlineRegex = re.compile('.\*')**

>>> **noNewlineRegex.search('Serve the public trust.\nProtect the innocent.\nUphold the law.').group()**

'Serve the public trust.'

>>> **newlineRegex = re.compile('.\*', re.DOTALL)**

>>> **newlineRegex.search('Serve the public trust.\nProtect the innocent.\nUphold the law.').group()**

'Serve the public trust.\nProtect the innocent.\nUphold the law.'



?匹配零次或一次前面的分组。

 \*匹配零次或多次前面的分组。

 +匹配一次或多次前面的分组。

 {n}匹配n 次前面的分组。

 {n,}匹配n 次或更多前面的分组。

 {,m}匹配零次到m 次前面的分组。

 {n,m}匹配至少n 次、至多m 次前面的分组。

 {n,m}?或\*?或+?对前面的分组进行非贪心匹配。

 ^spam 意味着字符串必须以spam 开始。

 spam$意味着字符串必须以spam 结束。

 .匹配所有字符，换行符除外。

 \d、\w 和\s 分别匹配数字、单词和空格。

 \D、\W 和\S 分别匹配出数字、单词和空格外的所有字符。

 [abc]匹配方括号内的任意字符（诸如a、b 或c）。

 [^abc]匹配不在方括号内的任意字符

**要让正则表达式不区分大小写，可以向re.compile()传入re.IGNORECASE 或re.I，作为第二个参数。**

>>> **robocop = re.compile(r'robocop', re.I)**

>>> **robocop.search('RoboCop is part man, part machine, all cop.').group()**

'RoboCop'

>>> **robocop.search('ROBOCOP protects the innocent.').group()**

'ROBOCOP'

>>> **robocop.search('Al, why does your programming book talk about robocop so much?').group()**

'robocop'

**你可以告诉re.compile()，忽略正则表达式字符串中的空白符和注释，从而缓解这一点。要实现这种详细模式，可以向re.compile()传入变量re.VERBOSE，作为第二个参数。**

phoneRegex = re.compile(r'((\d{3}|\(\d{3}\))?(\s|-|\.)?\d{3}(\s|-|\.)\d{4}(\s\*(ext|x|ext.)\s\*\d{2,5})?)')

**你可以将正则表达式放在多行中，并加上注释，像这样：**

phoneRegex = re.compile(r'''(

(\d{3}|\(\d{3}\))? # area code

(\s|-|\.)? # separator

\d{3} # first 3 digits

(\s|-|\.) # separator

\d{4} # last 4 digits

(\s\*(ext|x|ext.)\s\*\d{2,5})? # extension

)''', re.VERBOSE)